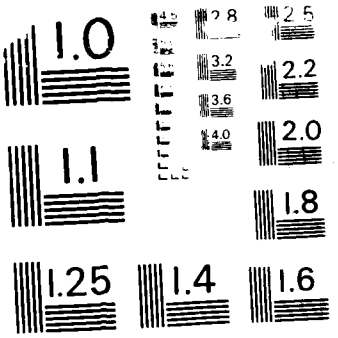
AD-A190 559 EXPRESSING MATHEMATICAL SUBROUTINES CONSTRUCTIVELY(U) 1/1
MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL
INTELLIGENCE LAB G L ROYLANCE NOV 87 AI-M-999
UNCLASSIFIED N00014-86-K-0180 F/G 12/5 NL



(4)



MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A. I. Memo No 999 November 1987

# Expressing Mathematical Subroutines Constructively

Gerald L. Roylance

**Abstract**

The typical subroutines that compute $\sin(x)$ and $\exp(x)$ bear little resemblance to our mathematical knowledge of these functions: they are composed of concrete arithmetic expressions that include many mysterious numerical constants. Instead of programming these subroutines conventionally, we can express their construction using symbolic ideas such as periodicity and Taylor series. Such an approach has many advantages: the code is closer to the mathematical basis of the function, less vulnerable to errors, and is trivially adaptable to various precisions.

**Acknowledgments.** This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency under Office of Naval Research contracts N00014-86-K-0180 and N00014-85-K-0124.

```
;;; calculate sin(z * (pi/2)), -1 <= z < 1
;;;   error < .5e-9
(define (sine-quad z)
  (let ((z2 (* z z)))
    (* z
       (+ 1.57079632662143
          (* z2
             (+ -0.64596409264401
                (* z2
                   (+ 0.07969258728630
                      (* z2
                         (+ -0.00468162023910
                            (* z2
                               (+ 0.00016021713430
                                  (* z2
                                     -0.00000341817225
                                     ))))))))))))))

(define (sine x)
  (let ((quad  (floor x (/ pi 2.0))))
    (let ((phase (- (mod (+ quad 1.0) 4.0) 1.0)))
      (if (> phase 1.0)        ;sin(pi+z)=-sin(z)
          (- (sine-quad (- phase 2.0)))
          (sine-quad phase))))
```

Figure 1: A Conventional Sine Routine in Lisp.

# 1 Introduction

Scientific subroutines such as $\sin(x)$ and $\exp(x)$ have few abstractions, are littered with numerical constants, and are tailored to specific machines. A Lisp translation of a typical sine subroutine is given in figure 1. What is it doing? Where do these multidigit constants come from? We should be suspicious of any code that looks like this; perhaps someone has miscalculated or mistyped one of the constants. The problem with this code lies in the poor description that numerical programmers use: they write down the results of a calculation (eg, 1.57079632662143) rather than what the calculation is (eg, first term of an economized Taylor series).

This poor description is unnecessary, and this paper provides an alternative called Constructive Programming (CP). Instead of writing a subroutine that computes the value of the sine function, the programmer writes code to construct the subroutine that computes a value. He essentially describes how he

would write the program rather than merely writing it. This level of indirection improves the clarity and the believability of the programs. A further benefit of CP is that a single description specifies a family of subroutines: just changing the number that specifies the accuracy will generate the single, double, or quadruple precision versions of a subroutine.

Constructive Programming can be as efficient as the conventional methods because the extra work of manipulating the constructive description is essentially carried out at "compile time". The meaning of "compile time" in a description based on higher order procedures is more general than when we run the compiler, and so CP can also produce efficient subroutines at "run time".

In the following sections, the sine routine will be rewritten in a constructive style. Initially the code will focus on describing the sine function and will ignore the issue of efficiency which is taken up in the later sections.

The procedures given here are in the Lisp dialect Scheme[1]. The actual code was written in Common Lisp[4], but its higher order procedures are syntactically more cumbersome.

## 2 Reducing the Interval

One of the most obvious things a sine routine does is exploit the periodicity of the sine function to map the sine's infinite domain onto a finite interval. The sine routine produces an approximation to the sine function, and approximations are most efficient if they cover only small intervals. Reducing the interval of approximation also eases the demand for high precision arithmetic.

Here is a definition of a sine routine that maps the argument into the interval $[-\pi/2, 3\pi/2]$ (this particular interval is used because it is symmetric about $\pi/2$ — a property that is important in a transform below). The routine **sine-full** is described below.

```
(define (sine-fcn x)
  (let ((cycles (floor (/ (+ x (/ pi 2)) (* 2 pi)))))
    (sine-full (- x (* cycles (* 2 pi))))))
```



Exploiting periodicity is something that we will do many times in the construction of mathematical subroutines, so it is worthwhile to encapsulate the idea of reducing an interval down to one period in a higher order procedure.

```
(define (period-maker fcn a period)
  (lambda (x)
    (let ((cycles (floor (- x a) period)))
      (fcn (- x (* cycles period))))))
```

This procedure takes a procedure **fcn** that is defined on [a, **a+period**] and returns a procedure that replicates that function over the rest of the real line. The code above is now achieved using **period-maker**:

```
(define sine-fcn
   (period-maker sine-full (- (/ pi 2)) (* 2 pi)))
```

A further reduction of the interval of approximation uses the reflective symmetry of the sine function about $\pi/2$.

```
(define (sine-full x)
  (sine-half (if (< x (/ pi 2)) x (- pi x)))))
```

As before, a higher order procedure will express this reflective symmetry.

```
(define (reflection-maker fcn a)
  (lambda (x)
     (fcn (if (< x a) x (- (* 2 a) x)))))
(define sine-full
  (reflection-maker sine-half (/ pi 2)))
```

While other trigonometric identities can be applied to further reduce the interval of approximation, they are usually not beneficial for reasons beyond the scope of this paper.

These procedures exploit properties of the sine function in order to reduce the routine's argument to a small interval (in this case, the domain of the as yet undefined procedure `sine-half`: $[-\pi/2, \pi/2]$). There are no obscure constants and we should feel comfortable with the trigonometric identities used. We must still address, however, the method of generating values of the sine over this reduced interval. The next section discusses the construction of a sine approximation on the half period.

# 3 Power Series Approximations

A transcendental function such as the sine function can be represented by a Taylor series:

$$\sin(x) = \sum_{i=0}^{\infty} (-1)^i \frac{1}{(2i+1)!} x^{2i+1}$$

This particular Taylor series is absolutely convergent for all values of $x$. Notice that there are no multidigit magic constants in the description of the Taylor series.

A simple way to procedurally represent the $i$th term of the sine power series is as a simple function:

```
(define (sine-term i)
  (term-make (/ (expt -1 i)
                (factorial (+ (* 2 i) 1))) ; coefficient
             (+ (* 2 i) 1)))               ; power
```

The function **(term-make a b)** produces a representation of $ax^b$. The **sine--term** procedure is a finite description of the infinite series. In the code that follows we will use this procedure as the representation of the sine function.

It would take infinite time to use all the terms of a power series, so we must have some method of truncating the series to a finite number of terms. The first $n$ terms of the power series can be turned into a polynomial (which we will call a **termlist**) with this code.

```
(define (sine-truncated-series n)
  (if (< n 0)
      (termlist-make)
      (termlist-adjoin (sine-term n)
                       (sine-truncated-series (1- n)))))
```

The procedure **termlist-make** creates a termlist with no terms; **termlist--adjoin** produces another termlist that has one more term. The general version that will truncate a power series represented by a term function is:

```
(define (truncated-series term-fcn n)
  (if (< n 0)
      (termlist-make)
      (termlist-adjoin (term-fcn n)
                       (truncated-series term-fcn (1- n))))))
```

Thus we can rephrase *the truncated sine series more simply* as

```
(define (sine-truncated-series n)
  (truncated-series sine-term n))
```

Now, with the aid of a function **termlist-eval** that evaluates the polynomial represented by a termlist for a particular point, we can approximate the sine function with our truncated series. For example, if we need only the first 10 terms of the series to get a required accuracy, then we could use this code to evaluate the sine function.

```
(define (sine-half x)
  (termlist-eval (sine-truncated-series 10) x))
```

While this routine will compute values of the sine function, it has a couple of severe problems: we don't know how accurate it is, and it is ridiculously slow and inefficient. The next sections will fix these problems without changing the basic strategy.

# 4 How Many Terms Should be Used?

The discussion in the previous section got rid of many magic numbers, but didn't eliminate all of them: the number of terms to include in the truncated

series is a magic number, so let's get rid of it. For a particular accuracy and a particular argument, we need to know the number of terms needed to attain that accuracy.

We should not truncate a series until we know that its terms are definitely getting smaller. Notice that the individual terms of the sine series

$$\sin(x) = \sum_{i=0}^{\infty} (-1)^i \frac{1}{(2i+1)!} x^{2i+1}$$

are monotonically decreasing when the denominator starts growing faster than the numerator. This point happens when $2i + 1 > x$.

The absolute error in the truncated sum of an alternating sign, absolutely convergent, series is less than the magnitude of the first term neglected. Thus we can write some code to tell us how many terms of the series we need to take. The function **term-eval** evaluates the term at a particular value of $x$.

```
(define (sine-number-of-terms eps x)
  (do ((i (ceiling (/ (- x 1) 2)) (1+ i)))
      ((< (abs (term-eval (sine-term i) x)) (abs eps))
       (1- i))))
(sine-number-of-terms 1.0e-6 (/ pi 2)) ->  5
(sine-number-of-terms 0.5e-9 (/ pi 2)) ->  7
```

Thus we only need 5 terms (which is a 9th degree polynomial) to find the sine of $\pi/2$ to 6 digits.

In order to generalize these ideas to other alternating sign, absolutely convergent, power series, we must specify when the absolute value of the series terms are monotonically decreasing and when the terms have are small enough to be ignored. When the series turns monotonic is, in general, a function of the series variable $x$, so we should use a lambda expression to specify it. For the sine example, we would use

```
(define (sine-mono x)
  (ceiling (/ (- x 1) 2)))
```

We only require that this function be conservative in its estimate.

With the aid of the monotonic function, we can determine the number of terms required to achieve an accuracy **eps**.

```
(define (number-of-terms term-fcn mono-fcn eps x)
  (do ((i (mono-fcn x) (1+ i)))
      ((< (abs (term-eval (term-fcn i) x)) (abs eps))
       (1- i))))
(number-of-terms sine-term sine-mono 1.0e-6 (/ pi 2)) ->  5
```

Rolling everything together, we can calculate an arbitrary (alternating sign, absolutely convergent) truncated power series that will achieve a desired accuracy.

```
(define (truncated-series-eps term-fcn mono-fcn eps x)
  (truncated-series term-fcn
                    (number-of-terms term-fcn
                                     mono-fcn
                                     eps x)))
```

Thus we can define a version of the half period sine routine using no magic numbers that is accurate to 9 digits.

```
(define sine-half-9
  (lambda (x)
    (termlist-eval
      (truncated-series-eps sine-term sine-mono
                            1.0e-9 (/ pi 2))
      x)))
```

We can, in principle, generate arbitrarily accurate routines this way. Using rational arithmetic (because our floating point numbers were not accurate enough!), we have generated sine routines that have 346 digits of precision.

# 5 Compile Time and Run Time

The unfortunate feature of the code we've shown so far is that every time we call the sine routine, we have to recompute the truncated power series and then evaluate it. That's a lot of overhead. It would be better if the truncated series could be determined once and never computed again.

A trivial change to the code above calculates the termlist only once — when the procedure is defined.

```
(define sine-half-9
  (let ((terms (truncated-series-eps sine-term sine-mono
                                     1.0e-9 (/ pi 2))))
    (lambda (x)
      (termlist-eval terms x))))
```

In this version, the termlist is explicitly calculated and then squirreled away in the environment of the closure (see figure 2) made by the lambda expression. Whenever the closure is called, it accesses the termlist stored in the environment instead of recomputing it.

Traditionally, this optimization is a compile time versus run time distinction. Doing things at compile time requires doing them only once; things done at run time are done again for each call. While intuitively this distinction is what we want, the world of higher order procedures is a little more complicated. In conventional programming languages, procedures are only built at compile time, but Lisp lets them be built at run time. The code above calculates the

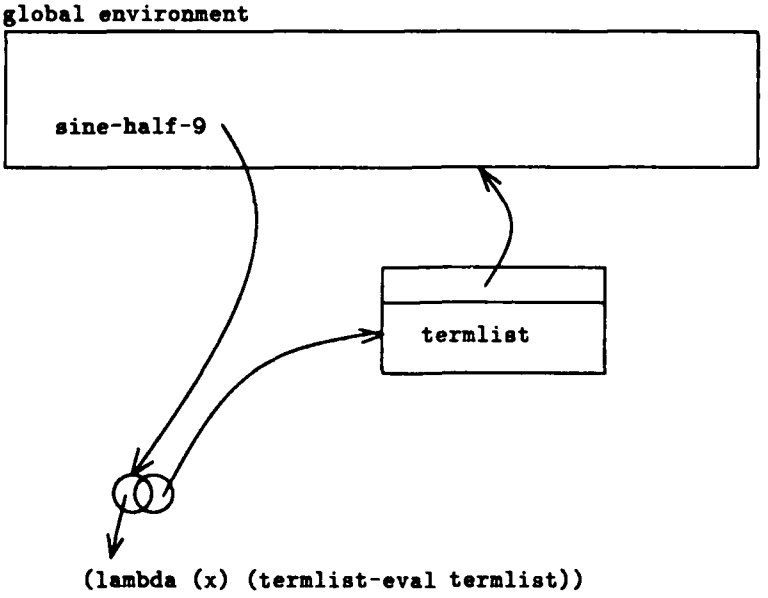



Figure 2: Environment Diagram for **sine-half-9**.

truncated series when the procedure (closure) is defined (definition time); it makes no difference whether the code is interpreted or compiled, the termlist will only be calculated once.

In theory, Lisp compilers could do extensive optimization of these procedures using transformations such as constant folding of the termlists and procedure integration of **termlist-eval**. In practice, Lisp compilers are not that advanced and there will be a performance penalty caused by lexical lookups, procedure call overhead, and the inability to compile out general type dispatches. These problems should be short-lived, but even if they are not, they have minimal impact on constructive programming. One can, for example, recast the above procedures as macros whose compile time expansions include these optimizations. Though that approach is expedient, it is not an appropriate long term methodology. Writing macros that do such optimizations also runs counter to the point of this paper: we want the programming language, not the programmer, to do the low level work.

# 6 Economization

By exploiting the periodicity and the symmetry of the sine function, we produced a sine routine that uses a finite number of the terms of the original Taylor

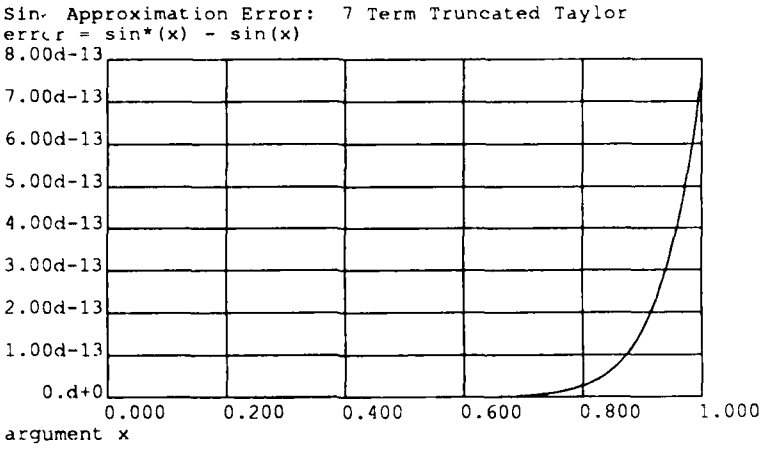



Figure 3: Uneconomized Sine Approximation

series. We could use this routine, but there are some other improvements that should be made. A Chebyshev economization [3] of the truncated series reduces the number of terms that need to be calculated. *Economization is a trick that is* worthwhile whenever a polynomial will be evaluated repeatedly (as in a subroutine library). The reduction in computation time can be significant: a 7 term polynomial can be reduced to 5 terms using a Chebyshev economization.

The reason that the truncated series for sine can be economized is that most of its error is concentrated near the end points of the approximation interval. Economization spreads this error throughout the interval. The initial error curve of a 6 term sine series is shown in figure 3. An economized error curve is shown in figure 4.

The expression

**(chebyshev-economization-scaled s poly initial-error error-bound)**

takes a polynomial (termlist) whose domain is **-s** to **s** and whose initial error is **initial-error** and returns a new polynomial whose error is less than **error-bound**. It is an error if **initial-error** > **error-bound**. The routine uses Chebyshev polynomials.

Figure 5 gives a definition of **sine-half-maker** that uses an *economized* series to produce a sine function to a desired accuracy.

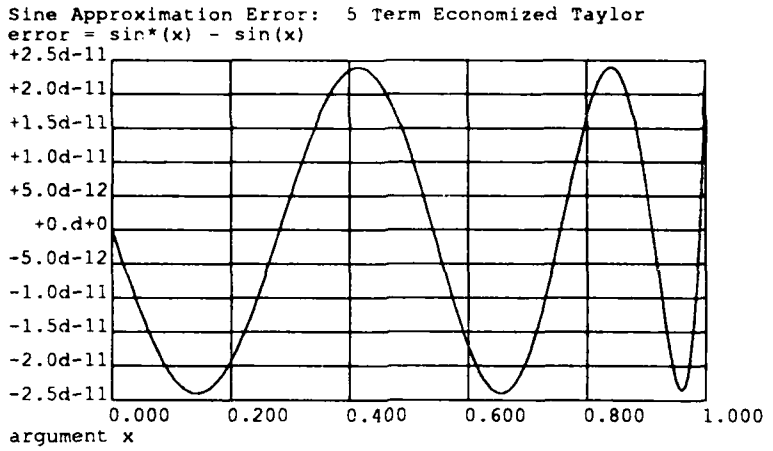



Figure 4: Economized Sine Approximation

```
(define (sine-half-maker eps)
  (let ((termlist
         (chebyshev-economization-scaled
           (/ pi 2)
           (truncated-series-eps sine-term sine-mono
                                 (/ eps 10) (/ pi 2))
           (/ eps 10)
           eps)))
    (lambda (x)
      (termlist-eval termlist x))))
```

Figure 5: **sine-half-maker**

# 7 Sine Summary

Programming a sine routine constructively involves specifying a few steps. The domain of the function is reduced to a reasonably small interval by employing the periodicity and symmetry of the sine function. Within that interval, the sine function is approximated by an economized Taylor series. In order to be readable and believable, a program should reflect these steps; the code in figure 6 tries.

# 8 Bessel Functions

Constructive Programming is not limited to sine routines; it applies to a broad class of problems that occur in subroutine libraries and even includes some exotic functions. For example, these methods can be used on Bessel functions. Abramowitz[2] (section 9.1.10) gives the power series expansion for the Bessel functions of integer order $\nu$ as:

$$J_\nu(z) = (z/2)^\nu \sum_{k=0}^{\infty} \frac{(-1/4z^2)^k}{k!\Gamma(\nu + k + 1)}$$

Translating this expression into the termlist representation (and making use of $\Gamma(n+1) = n!$ for integer $n$ gives us a **term-maker** function:

```
(define (bessel-term-maker v)
  (lambda (k)
    (term-make (* (expt (/ 1 2) v)
                  (/ (expt (/ -1 4) k)
                     (factorial k)
                     (factorial (+ v k))))
               (+ v (* 2 k)))))
```

A term-maker function is used here because the Bessel functions are a parameterized family. Calling **bessel-term-maker** with an argument **v** gives us a term function for that order Bessel function.

The Bessel function series is monotonic decreasing if $|z|/2 < k$, so we can determine the term after which the terms are monotonically decreasing:

```
(define (bessel-mono-maker v)
  (lambda (x)
    (ceiling (/ (abs x) 2))))
```

These two functions now let us construct a termlist to arbitrary accuracy. **bessel-series-eps** returns a termlist for the Bessel function of order **v** that has accuracy **eps** as long as the argument does not exceed the value **x**.

```
;;; Describe the Taylor Series
(define (sine-term i)
  (term-make (/ (expt -1 i)
                (factorial (+ (* 2 i) 1)))
             (+ (* 2 i) 1)))
(define (sine-mono x)
  (ceiling (/ (- x 1) 2)))

;;; Build a half period sine to any accuracy
(define (sine-half-maker eps)
  (let ((termlist
         (chebyshev-economization-scaled
           (/ pi 2)
           (truncated-series-eps sine-term sine-mono
                                 (/ eps 10) (/ pi 2))
           (/ eps 10)
           eps)))
    (lambda (x)
      (termlist-eval termlist x))))

;;; Expand a half period sine to the real line
(define (sine-maker eps)
  (let* ((sine-half (sine-half-maker eps))
         (sine-full (reflection-maker sine-half (/ pi 2))))
    (period-maker sine-full (- (/ pi 2)) (* 2 pi))))

;;; Make an instance accurate to 9 digits
(define sine
  (sine-maker 0.5e-9))
```

Figure 6: Sine Routine Without Magic

```
(define bessel-0-8-c
  (let* ((a        3.0d0)
         (eps      5.0d-8)
         (eps2     (/ eps 1000))
         (termlist (chebyshev-economization-scaled
                     a
                     (bessel-series-eps v a eps2)
                     eps
                     eps2)))
    (lambda (x)
      (termlist-eval termlist x))))
```

Figure 7: `bessel-0-8-c`

```
(define (bessel-series-eps v x eps)
  (truncated-series-eps (bessel-term-maker v)
                        (bessel-mono-maker v)
                        eps x))
```

Chebyshev economization is also appropriate here and also little trouble. Figure 7 has an economized Bessel procedure that is accurate to 7 digits. A plot of the approximation error is given in figure 8.

Earlier we were suspicious about code that used magic numbers. Abramowitz, in section 9.4.1, gives the coefficient values for a Bessel function approximation over the interval $[-3, 3]$ with an accuracy of 5.0E-8. A plot of the approximation error is given in Figure 9. The approximation that we computed (without magic numbers) has just as many terms as in the Abramowitz reference, **but it is 40 times more accurate**. The approximation they give is not wrong because they correctly state the error bound, but it is sad that they needlessly threw away some available accuracy.

# 9 Summary

This paper has only shown a few techniques for approximating functions, but there are many others. The primary goal has been to show how to express code in terms of its development rather than as a sequence of arithmetic operations. Such an expression makes explicit the properties of the approximation - its accuracy, for example, is an integral part of the code rather than a comment that the programmer just happened to add to the source code amid several mysterious numbers. A further benefit of this approach is that one expression of a function will provide single, double, and quadruple precision instances.

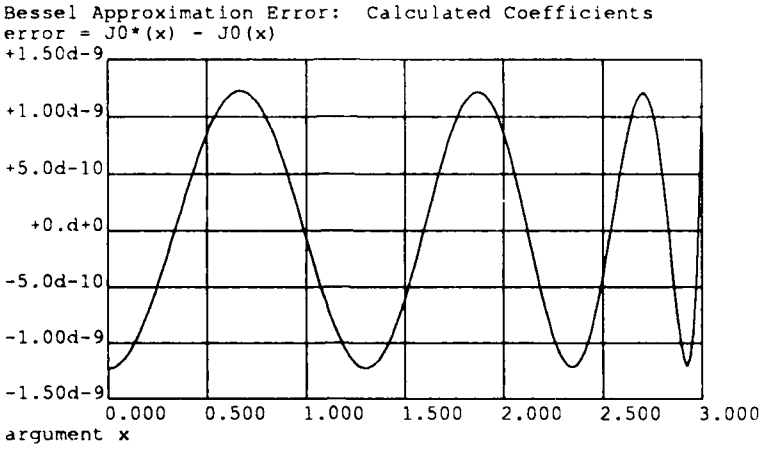



Figure 8: Error curve for economized Bessel function.

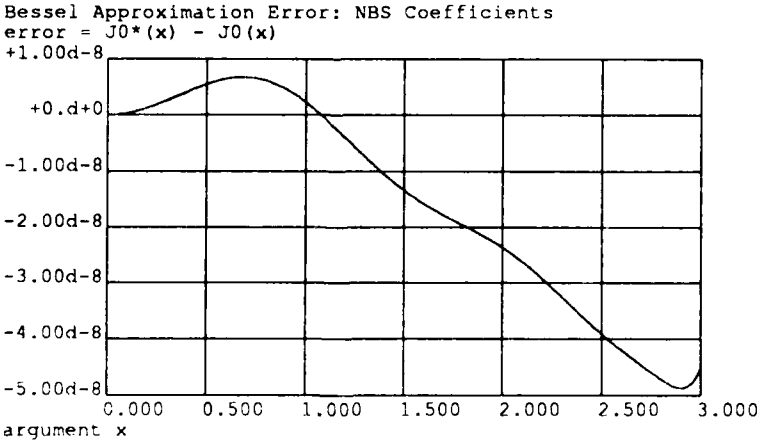



Figure 9: Error curve for NBS Bessel function

The paper also argued that run time efficiency does not suffer. While the construction of a function does require doing a mathematical derivation, that derivation need only be done once. The actual calls of the function are not encumbered.

There is, however, still a lot of magic in the descriptions given here. The code has implicitly assumed that series are alternating sign and absolutely convergent, but conventional routines make these same assumptions. There are other problems that the above discussion did not address: desired error metric (eg, absolute error or relative error), more detailed descriptions of algorithm restrictions (eg, argument ranges and argument types), and the accuracy of the arithmetic.

# References


[1] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, 1985.

[2] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* U. S. Department of Commerce, National Bureau of Standards, Washington DC. 1972.

[3] Forman S. Acton. *Numerical Methods That Work.* Harper & Row, New York. 1970.

[4] Guy L. Steele Jr. *Common Lisp.* Digital Press, 1984.

# REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

| | |
|---|---|
| 1. REPORT NUMBER: AI Memo 999 | 2. GOVT ACCESSION NO.: ADA190559 |
| 3. RECIPIENT'S CATALOG NUMBER: | |
| 4. TITLE (and Subtitle): Expressing Mathematical Subroutines Constructively | 5. TYPE OF REPORT & PERIOD COVERED: memorandum |
| | 6. PERFORMING ORG. REPORT NUMBER: |
| 7. AUTHOR(s): Gerald Roylance | 8. CONTRACT OR GRANT NUMBER(s): N00014-86-K-0180 N0014-85-K-0124 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS: Artificial Inteligence Laboratory 545 Technology Square Cambridge, MA 02139 | 10. PROGRAM ELEMENT PROJECT, TASK AREA & WORK UNIT NUMBERS: |
| 11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209 | 12. REPORT DATE: November, 1987 |
| | 13. NUMBER OF PAGES: 15 |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Office of Naval Research Information Systems Arlington, VA 22217 | 15. SECURITY CLASS. (of this report): |
| | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: |

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Mathematical Subroutines

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The typical subroutines that compute sin(x) and exp(x) bear little resemblance to our mathematical knowledge of these functions: they are composed of concrete arithmetic expressions that include many mysterious numerical constants. Instead of programming these subroutines conventionally, we can express their construction using symbolic ideas sych as periodicity and Taylor series. Such an approach has many advantages: the code is closer to the mathematical basis of the functions, less vulnerable to errors, and is trivially adaptable to various precisions.

END

DATE

FILMED

4-88

DTIC